## Trust Management and Security

This document gives an overview on the Hyperty Trust Model as well as on Hyperty Sandbox runtime execution environment.

Hyperties are securely associated to User Identities selected by the end-user himself. Hyperty Users are human beings (including group of human beings e.g. corporation) or things (including group of things and physical spaces e.g. a smart home or smart building).

Hyperty Trust Model extends [WebRTC Identity model](https://w3c.github.io/webrtc-pc/#sec.identity-proxy) where Identity tokens are generated, inserted in intercepted Messages sent by Hyperties and validated by recipient Hyperty Runtimes before delivered to the target Identity. These identity management procedures are performed according to applicable policies managed by the end-user.



Hyperty Trust Management

### User Identity

In our modern society, technology is ubiquitous, and transactions are evermore accomplished using digital technologies without the need to involve physical contact. An example of this situation can be observed in money transactions, whilst a few years ago if someone needed to make a bank transfer, it would require that person to move personally into a bank agency to order it, and in current days these money transfers can be performed using a smartphone. To achieve this, we need digital credentials to prove who we are and what we are allowed to do in remote communication. Therefore two important information security mechanisms must be implemented: authentication and authorization.

Authentication is verifying the identity claimed by an actor. Usually, to authenticate users, credentials make use of one or several factors among:

* something a user knows (such as a PIN code or a password),
* something a user owns (such as a SIM card),
* something a user is (such as a fingerprint, or a voice sample),
* something a user does (such as typing characteristics).

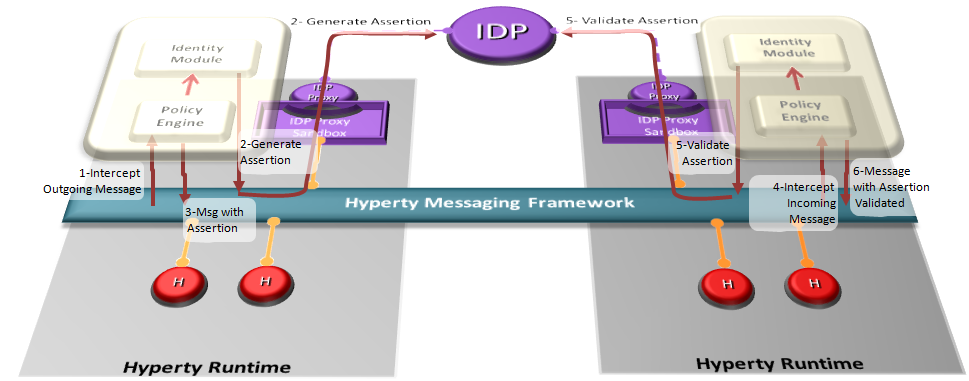
Authorization is deciding whether a given identity may execute or access a certain resource. Access control to a service or system, can be achieved based on access rights or policies that allow or deny a particular action based on an identifier, a role (RBAC), or an attribute (ABAC).

Hyperty trust model relies on service-independent authentication that is global and non-service-bound. In implementations, each Hyperty service provider may include their own user recognition (i.e. their own internal user accounts) and service authorization (i.e. level of permissions to use the service), over and above the initial user identification.

#### Identity Module and IdP Proxy

The Identity Module (Id Module) is the [Core Runtime](https://github.com/reTHINK-project/dev-runtime-core) component responsible for handling the user identity and the association of this identity with the Hyperty instances. The identity in the reTHINK project is not fixed to a unique Identity source (espically it is not bound to a single Service Provider), but obtained through several Identity P£rovider. In this approach, the Id Module provides the user the option to choose the preferred IdP for authentication. For example, a user with a Google account can use Google as an Identity Provider to retrieve Identity Tokens that may be associated by the Identity Module with a Hyperty instance. Following the flexible and adaptable approach fostered by WebRTC standards (from both [IETF](https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-12#section-5.6.2) and [W3C](https://www.w3.org/TR/webrtc/#sec.identity-proxy)) IdP Proxies are downloaded from each Identity Provider to act as intermediaries between the Identity Module and the aforementioned Identity Providers. The IdP Proxy is the [IdP ProtoStub](hyperty-messaging-framework.md#protocol-on-the-fly-protofly-and-protostubs) that is responsible to handle the communication between the Identity Module and a specific Identity Provider.

The following figure illustrates this interaction:



Interaction between the Identity Module and the Identity Provider

*NB. The Policy Engine is the Core Runtime component rsponsible to manage and apply policies to perform access control. More details are found in the* [*Policy Management*](https://github.com/reTHINK-project/specs/tree/master/policy-management) *section.*

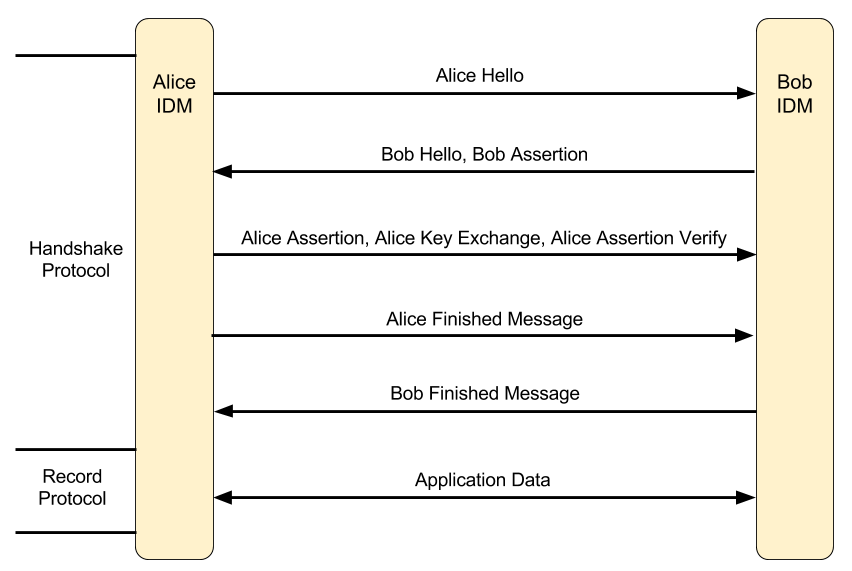
#### Mutual Authentication

Identity management relies upon managing life-cycle of identity-related security tokens. Therefore, identity assertions are used in reTHINK to identify a user and to prove that he was authenticated on an IdP. The IdP asserts a particular content for the user, provided during the request for authentication by the IdM, after a successful authentication this same user.

reTHINK architecture is designed to operate in a peer-to-peer architecture, and as a result there is no centralised service that proceeds to authenticate the users in reTHINK. Because of that, the identity assertions play a very important role in reTHINK, by enabling mutual authentication between users: here agin reTHINK promotes ideas similar to [WebRTC working group](https://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-12). Using IdP protocols like [OpenID Connect](http://openid.net/developers/specs/) it is indeed possible to request the IdP to assert a particular content on the request to authenticate the user. The identity token received after the user's successful authentication contains the user identity assertion and the content provided during the authentication request, also asserted. In reTHINK, the content to be asserted by the IdP is a public key specified by the Identity Module, later used to prove the user's identity to a third party. This way, the identity assertion provided by the IdP acts as a digital certificate, where the IdP plays the role of a Certification Authority (CA).

As such, whenever a user intends to initiate a communication with another user, the mutual authentication protocol is triggered so that users can authenticate each other mutually. In order for the mutual authentication to be successful, all the messages are required to have an identity assertion, which will work as a digital certificate. Therefore, to authenticate to a message, the sender identity assertion obtained through the IdPs is attached to the message, containing the user public key. To confirm that the public key actually corresponds to the claimed identity, the receiving user contacts the IdP to validate the content of the Identity assertion. With this public key, the receiver can validate the sender message digital signature and encrypt the response to the sender challenge to conclude and successfully authenticate the sender identity. For mutual authentication, the roles invert and the receiving users becomes the one who must prove his identity, using the same procedure. For user privacy assurance, the Identity Module may frequently request the generation of a new Identity assertions, each with different public keys.

Taking the mutual authentication process described above, the IdM performs this authentication process and along with it, generates the symmetric session keys to be used for the protection of the messages exchanged after the mutual authentication process. This is done in an identical manner as in the TLS protocol. By doing this, we provide the same procedures and the same security properties of the TLS, including the security assurance for the message integrity and confidentiality. The proposed solution used in rethink for authentication is also separated in the handshake and record phases. Some simplifications are introduced in the authentication protocol compared to TLS, mainly because there is no need to support some of the features. The negotiation steps of cryptographic methods in TLS are not taken in consideration since it is the Identity Module that defines the cryptographic methods to be used, and all devices running the reTHINK application will use the same version of the Identity Module. Additionally, compression will not be considered, since this protocol is already running on top another TLS communication (i.e. the communication of each client with the Message Node).



reTHINK mutual authentication flow

The figure above illustrates all the messages exchanged to provide mutual authentication between the users Alice and Bob. The following provides an explanation of the overview on the developed mutual authentication flow.

* **Alice Hello**: This message contains a number generated randomly by Alice.
* **Bob Hello, Bob Assertion**: This message contains a random number generated randomly by Bob and his identity assertion with his public key.
* **Alice Assertion, Alice Key Exchange, Alice Assertion Verify**: In case Alice is not in anonymous mode, she sends her identity assertion with her public key and sends a randomly generated premaster key encrypted with Bob's public key. In case Alice identity assertion is sent, the assertion verification must also be sent in the message to prove the ownership of the public key. This verification consists in the signature of the hash of all the previous messages exchanged and the content of this message.
* In this phase both Alice and Bob generate a master key using the premaster key and both random values exchanged in the beginning. With the master key the following keys are computed: Alice's MAC Key, Bob's MAC Key, Alice's encryption Key, Bob's encryption Key.
* **Alice Finished Message**: This is the first message to use the keys generated previously, and contains the MAC from the result of a pseudo random function that receives the master secret and all the handshakes messages exchanged previously as argument. This result is also encrypted with the Alice's encryption key.
* **Bob Finished Message**: This message is the response to the Alice finished message. It uses the keys generated previously to generate a MAC and encrypt that value. The MAC is obtained from the result of a pseudo random function that receives the master secret and all the handshakes messages exchanged previously as argument. This result is also encrypted with the Bob's encryption key.
* In this phase the authentication or mutual authentication is complete with all the necessary keys generated. This symmetric key allows for a secure communication between two users.
* **Application Data**: This message contains the body of the message encrypted with the encryption key of the sender to grant confidentiality, and a MAC of the entire message with the sender's MAC key to grant integrity.

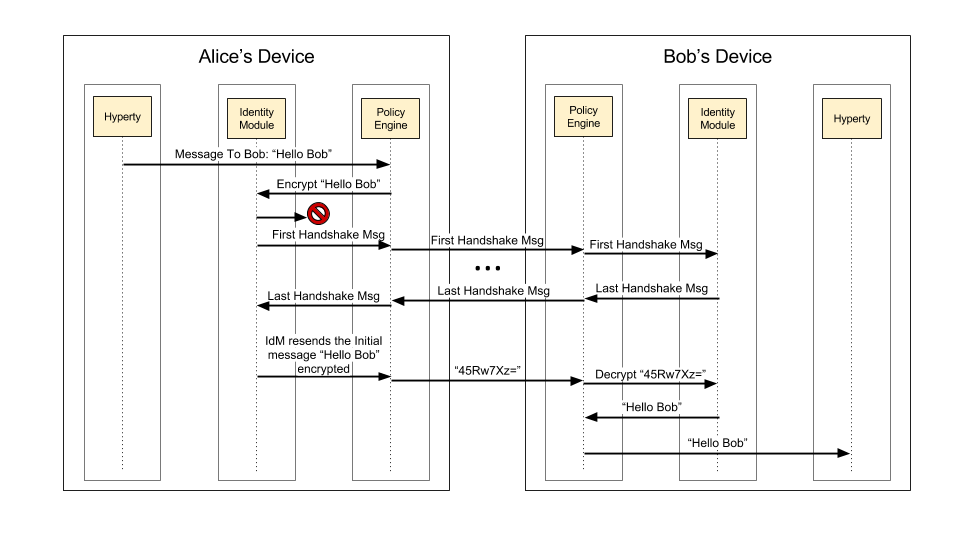
*NB. Despite the emphasis of the protocol developed for the IdM being in the mutual authentication it also supports anonymous communication, where the user who initiates the call starts it in anonymous mode. The only difference in the protocol flow is that the user who starts the communications does not send his identity assertion, not allowing the user who receives the request for communication to verify the caller's identity. The decision of accepting or not the anonymous communication rests with the policy engine.*

##### When does the mutual authentication occur?

The mutual authentication protocol is mandatory so that a secure communication channel can be established. Whenever some user intends to start a communication with another user the mutual authentication protocol must be triggered. The mutual authentication can start in two situations: when a message from a Hyperty URL to a Hyperty URL is sent for the first time between those Hyperties, and when an IdM method is called to explicitly start a mutual authentication between two Hyperties URLs.

The IdM method that triggers the mutual authentication protocol is called in a specific case, being when the Policy Engine receives a request to accept a subscription to a group chat, or when the current user invites another user to subscribe his Data Object.

Regarding the activation triggered by sending a message for a Hyperty for the first time, the Figure below illustrates the message flow for the first message sent between two Hyperties, one from Alice and one from Bob sent via the Message Bus. Only the essential components to demonstrate the flow are illustrated, which in this case are the Hyperty, the Identity Module (IdM) and the Policy Engine (PE). It is assumed that all messages sent via the Message Bus are intercepted by the Policy Engine, in order to be filtered according to the defined policies.



Example of Alice sending a message to Bob, for the first time

Taking data flow as an example, Alice starts by sending for the first time, the message "Hello Bob" to Bob. The PE intercepts the message and sends it to the IdM, to encrypt the message. The IdM tries to find the cryptographic keys generated for the secure communication between Alice and Bob. Since it is the first communication between Alice and Bob no cryptographic keys exists and the identity of Bob has not yet been authenticated. In this case, the IdM suspends the transmission of that message and starts immediately the mutual authentication protocol.

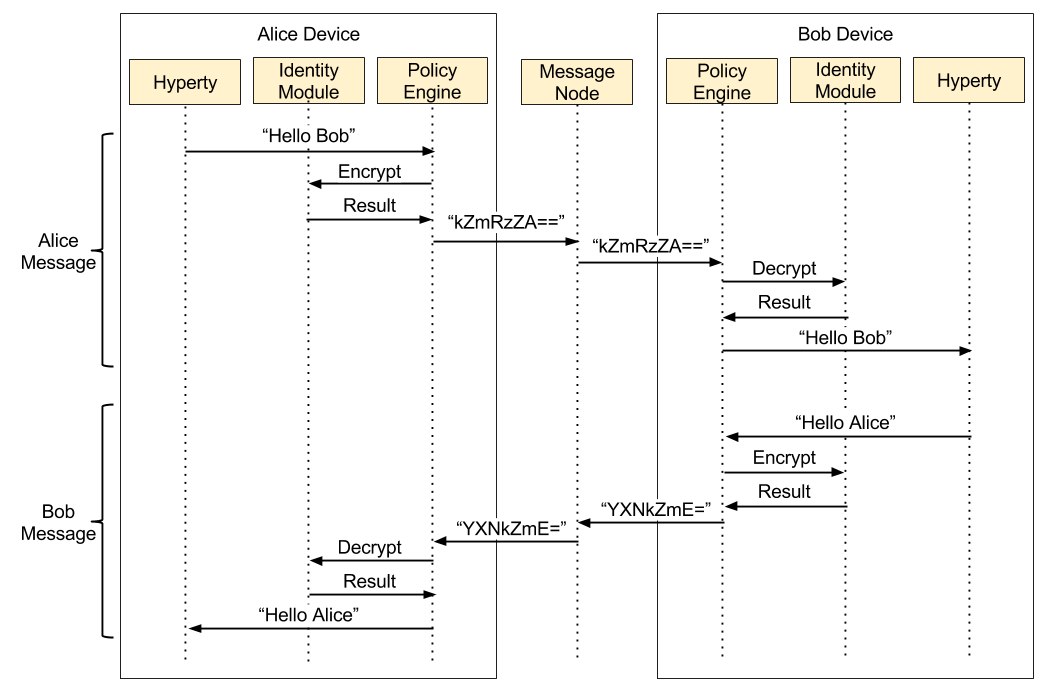
After receiving the last message of the handshake protocol and if Bob's identity is verified, Alice's IdM unblocks the initial message that triggered the mutual authentication protocol. However, the message is now sent encrypted with her symmetric key and with the integrity of the message protected by adding the hash of the message. This message is forward to the Bob's device and is intercepted by the PE, which requests Bob's IdM to decrypt the message and verify the hash. The IdM then uses Alice's symmetric keys to decrypt and verify the hash of the message, and passes the message with the original value "Hello Bob" sent by Alice. To conclude, the PE receives the message, decrypted and validated, and forwards it to the destination Hyperty.

#### Secure Communication

The communication between users is one of the major characteristic in reTHINK. As such, the authentication of each user takes a big role to ensure that no personification can occur. Every time a user starts a communication with another user, the process of mutual authentication, described above, is initiated by the Identity Module. This mutual authentication is not only useful for the authentication of the users, but is also essential for the exchange of the symmetric keys used in the established secure communications. In case, one of the users starts a communication anonymously, the authentication of the other user is made, so it can be possible to establish a secure channel between the two users.

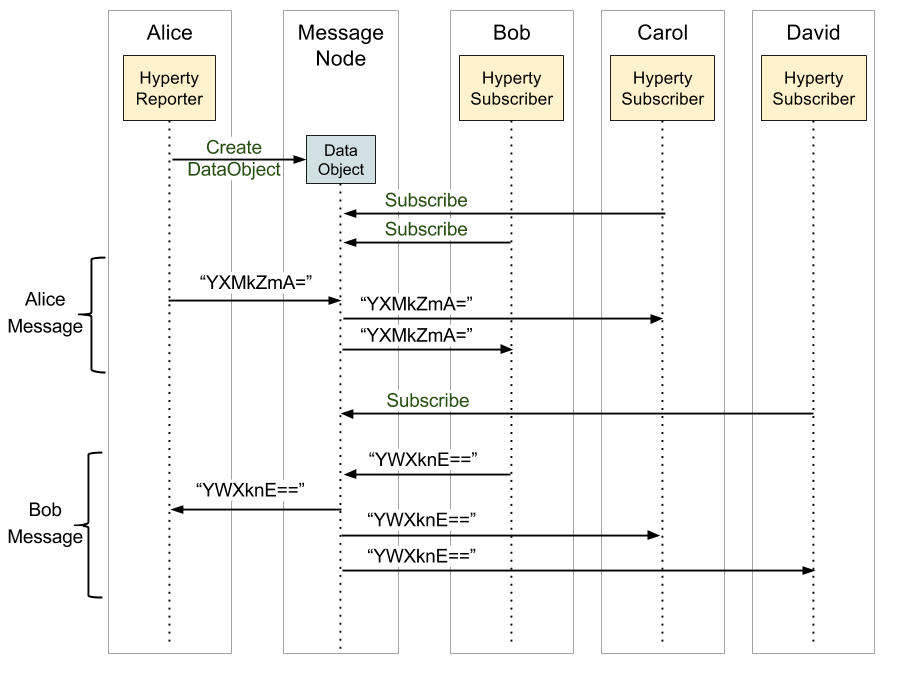
The reTHINK framework provides support for two types of communication, a direct communication between two users through their respective Hyperties and a group chat communication, using Data Objects, where the messages are exchanged between all participants in the group, subscribed to that Data Object. The encryption of these communications is optional, depending on the user preferences regarding secure channels. This secure channel consists on the creation of a HMAC of the message followed by the encryption of the message, to ensure confidentiality and integrity.

To implement the secure channel on a direct communication between two Hyperties, the Identity Module needs to capture these messages, in order to secure the contents of the messages exchanged. Using Figure 5 as an example of a secure communication between Alice and Bob, when Alice sends a message through her Hyperty, this message is intercepted by the Policy Engine, since all messages passe by it. The Policy Engine sends that message to the Identity Module, to be encrypted with Alice's session key and authenticated with the Alice MAC's key, with these two keys generated during the mutual authentication process. After the message manipulation, the Identity Module returns the message to the Policy Engine, to be sent to the public Message Node. The Message Node then forwards the message to the Bob device, where the Policy Engine running in the Bob device, intercepts it and applies the same steps used in the protection phase, but in this time to decrypt the message. After the Policy Engine receives the decrypted message by the Identity Module, returns it to the Hyperty of Bob in plaintext. When Bob sends a message, the same flow is used, with the only difference being in the key used to encrypt and to authenticate, which in this case uses the set keys of Bob.



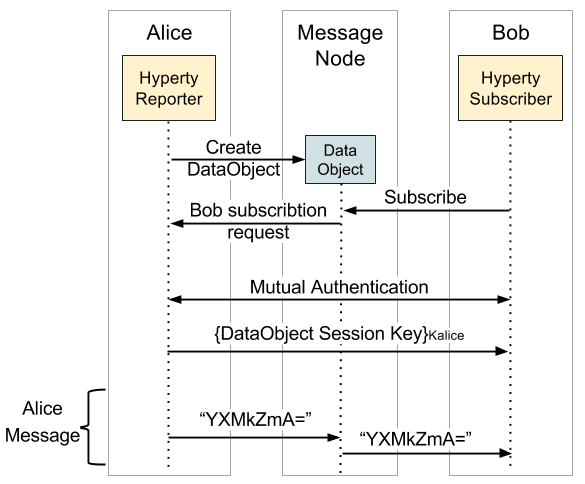
Hyperty to Hyperty communication

In a group chat communication, the communication starts with the creation of a Data Object by the Hyperty Reporter and the subscription from other Hyperties to that Data Object. After that, when a Hyperty sends a message, it is broadcasted to all Hyperties subscribed. The encryption of these messages is optional, but in case of opting to use a secure channel, these messages are encrypted with a symmetric key shared by all participants in the group chat.



Group chat communication

The Hyperty reporter, in a group chat, is responsible for managing the session key and the authentication of others Hyperties that join that chat group. Following the creation of a Data Object by the reporter, he generates a symmetric key and associates it to the Data Object. As illustrated in the Figure below, after the creation of the Data Object, when a Hyperty makes a request for subscription, this request is forward to the reporter, which starts the mutual authentication between the reporter and that subscriber. When the authentication is completed with success, the reporter encrypts the symmetric key associated to the Data Object with the reporter session key, obtained through the mutual authentication process and shared by both Hyperties reporter and subscriber. After that, the subscriber has the key which allows him to decrypt all messages exchanged. In the end, all Hyperties that successfully subscribe the same Data Object will end with the same symmetric key for that session.



Hyperty subscription flow

#### Identity selection

The Identity Module is responsible to obtain, manage the identities in reTHINK, and provide the identities when the Runtime Registry requires an identity to associate with a Hyperty. For the users to be able to select the desired identity, a GUI is mandatory. An identity GUI is part of the reTHINK web browser application, and allows the user the act of registering a Hyperty to select the identity he wants to associate with it.

The identity GUI can be presented in two distinct ways: by clicking in the settings button on the reTHINK web application, or by clicking in the button to register a Hyperty that will trigger the appearance of the identity GUI. This GUI allows the user to add new identities from a list of IdPs, choose an identity from the existing ones and remove identities from the list of previously registered identities. For example, when a user using the reTHINK web application click the button to register a new Hyperty the identity GUI appears listing all identities previously registered, and the option to add new ones. After the user selects the desired identity or chooses to obtain a new one, the Hyperty registration process continues.

### Runtime Sandbox

The hyperty runtime implements sandboxing mechanisms that ensure the correct isolation of client JavaScript code (i.e., Hyperties, ProtoStubs, and Applications). Isolation means that client code is confined to execute within the address space of an independent sandbox. As a result, sandboxes prevent potentially malicious code from interfering with client code instances in co-located sandboxes or from accessing external resources in the surrounding environment (e.g., files, network, etc.). Communication outside the sandbox is possible through well defined channels. Both sandboxing mechanisms and communication channels implemented by the hyperty runtime are available to the application programmer throught specific APIs and are dependent on the targeted platform.

For the browser platform, sandboxing is enforced by leveraging native mechanisms provided by the browser API. The core runtime components execute inside an iFrame. The iFrame implements the core sandbox by isolating the code of the core runtime from the main window in which the application Javascript code is executed. Application code is therefore prevented from accessing directly to the memory address space of the core runtime. Communication between application and core runtime is possible only through a single and well defined entrypoint which allows them to exchange messages: method postMessage(). Hyperties and protoStub downloaded from different domains are executed in independent Web Workers. Web Workers effectively isolate their internal states from each other and from the core runtime. The postMessage() method constitutes the only communication bridge between the these components.

For the standalone platform, the sandboxing mechanisms we employ are similar to the browser platform. The main difference is that, instead of using a browser, we leverage Crosswalk to support standalone applications. Crosswalk is an HTML application runtime that allows us to execute the hyperty runtime as native mobile applications in Android and iOS devices without the need to install a full-blown browser. Mobile applications only need to be bundled with both Crosswalk webviews and the hyperty runtime code. Since a Crosswalk webview implements a Chromium-based runtime, it can seamlessly execute the hyperty runtime code that was implemented for the browser platform.